Model-Driven User Interface Generation and Adaptation in Process-Aware Information Systems

Jens Kolb, Paul Hübner and Manfred Reichert

Institute of Databases and Information Systems
Ulm University, Germany
{jens.kolb,paul.huebner,manfred.reichert}@uni-ulm.de
http://www.uni-ulm.de/dbis

Abstract. The increasing adoption of process-aware information systems (PAISs) has resulted in a large number of implemented business processes. To react on changing needs, companies should be able to quickly adapt these process implementations if required. Current PAISs, however, only provide mechanisms to evolve the schema of a process model, but do not allow for the automated creation and adaptation of their user interfaces (UIs). The latter may have a complex logic and comprise, for example, conditional elements or database queries. Creating and evolving the UI components of a PAIS manually is a tedious and error-prone task. This technical report introduces a set of patterns for transforming fragments of a business process model, whose activities are performed by the same user role, to UI components of the PAIS. In particular, UI logic can be expressed using the same notation as for process modeling. Furthermore, a transformation method is introduced, which applies these patterns to automatically derive UI components from a process model by establishing a bidirectional mapping between process model and UI. This mapping allows propagating UI changes to the process model and vice versa. Overall, our approach enables process designers to rapidly develop and update complex UI components in PAISs.

1 Introduction

Process-aware information systems (PAISs) separate process execution from application code, i.e., the implementation of process activities. Hence, separation of concerns, which is a well-established principle in computer science is realized based on explicit *process models*. When initially capturing business processes in process models, focus is put on business aspects, while technical aspects concerning later process execution are excluded. Usually, the resulting business process models cover the users' activities at a fine-grained level (cf. Figure 1a). Hence, before deploying such a business process model in the PAIS, therefore, it must be revised and customized. For example, several human tasks, forming a process fragment in the business process model, may have to be combined into one activity in the executable process model (cf. Figure 1b). This activity is then

implemented by a respective *user interface (UI) component* in the PAIS, e.g., a user form whose corresponds to the one of the initial process fragment (cf. Figure 1c). Based on such a logic, for example, form elements may be disabled when selecting a certain check box, or database queries or web services may be run in the background to fetch or save data. Overall, both the implementation and maintenance of the UI components in a PAIS is a cumbersome and costly task. In particular, this hinders quick adaptations of process implementations [1].

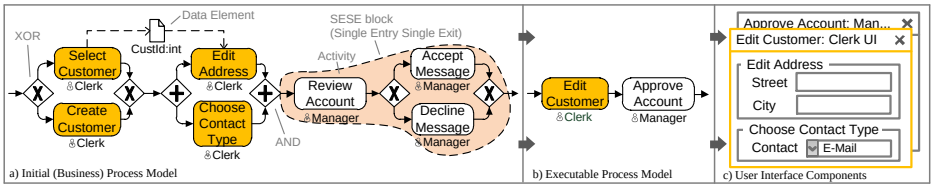


Fig. 1. Deriving UI Components from a Business Process Model

The proper evolution of the processes implemented in a PAIS is a critical success factor for any company [2]. It not only requires changes of the process models, but of the associated UI components and their internal logic as well (e.g., adding new input fields). Process model evolution is a well-understood feature in modern PAIS [3,4]. There exist editors for defining and changing simple UI components of the PAISs (e.g., moving or renaming input fields in a user form). However, complex changes of the internal logic of user forms can not be accomplished by users, but require process implementers to intervene. Moreover, the automatic propagation of changes made in the UI components to business process model and vice versa is not supported.

This paper addresses these issues through the automatic generation of complex UI components out of process model fragments. In this context, we first present common patterns for transforming process fragments to UI components. Thereby, we distinguish between *elementary* and *complex transformations*. While *elementary transformation patterns (ETP)* transform single activities of a process model to simple UI elements, *complex transformation patterns (CTP)* enable the mapping of entire process fragments and their logic to UI components, showing the same behaviour as the original process fragment does. Additionally, we provide an advanced transformation method for business process models that allows generating sophisticated UI components out of these models based on the user roles assigned to process activities. This method further allows propagating changes of UI components to the corresponding process model and vice versa. Especially, for human-centric processes, our transformation method decreases the effort for evolving PAIS to accommodate to changing needs.

The paper is structured as follows: Section 2 introduces basic notions. Section 3 describes common patterns for transforming process model fragments to UI components. Section 4 then presents a method for transforming process models

to a set of UI components. This method is based on transformation patterns and role-based process views. Section 5 sketches our proof-of-concept prototype. Section 6 discusses related work and Section 7 summarizes the paper.

2 Basic Notions

A process model is described in terms of a directed graph whose node set comprises *activities*, *gateways*, and *data elements*. An activity either corresponds to a *human task* and thus requires user interactions, or to a *service* representing an *automated task*. In turn, gateways can be categorized into *AND*, *XOR* and *Loop* and are used for modeling parallel/conditional branchings and loop structures. Edges between activities and/or gateways represent precedence relations, i.e., the *control flow* of the process model (cf. Figure 1a). Furthermore, *data elements* comprise *primitive* data elements and *complex* ones. Primitive data elements cover elementary data values of the process model and have one of the following types: *integer*, *float*, *boolean*, *string*, *date*, or *URI*. In turn, complex data elements compose primitive and/or complex data elements. Based on this, the data flow is defined by a set of directed edges connecting data elements and activities. *Writing* a data element is expressed through an edge pointing from an activity to the data element. In turn, *reading* a data element is expressed through an edge from this data element to the activity. We presume that process models are *well-structured* [5,6], i.e., sequences, branchings (of different semantics), and loops are specified as blocks with well-defined start and end nodes having the same gateway type. These blocks, also known as *SESE (single-entry-single-exit)* blocks, may be nested, but are not allowed to overlap (cf. Figure 1a).

3 User Interface Transformation Patterns

This section introduces a set of well-elaborated patterns for transforming a process fragment to a corresponding UI component. Section 3.1 discusses how we identified these *UI transformation patterns*. Section 3.2 then introduces *Elementary Transformation Patterns* (ETP), which constitute the basis for the *Complex Transformation Patterns* (CTP) presented in Section 3.3.

3.1 Pattern Identification

The goal of our research is to *identify and apply a general set of UI transformation patterns, which allow mapping process fragments to UI components*. To achieve this goal, we first describe a three-step method, which we apply for identifying relevant UI transformation patterns (cf. Figure 2). In Step 1, we analyze and evaluate PAIS engineering projects in which we were involved in the past. More precisely, we analyze the business process models from these projects as well as their technical implementation and related UI components. In Step 2,

we analyze existing PAISs (e.g., IBM WebSphere Lombardi, AristaFlow BPM Suite [7]) and their support for UI generation. Finally, in Step 3 we scan existing literature on model-driven development (MDD) of UI components [8,9,10].

The empirical results of these three steps are used to specify general UI transformation patterns. These either transform single process model elements, like activities and their inputs/outputs, to simple UI elements (i.e., *elementary transformation patterns (ETP)*) or entire process fragments to complex UI components (i.e., *complex transformation patterns (CTP)*).

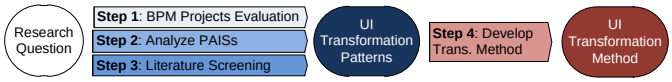


Fig. 2. Transformation Pattern Identification Method

Based on the identified transformation patterns, Step 4 develops a transformation method allowing for the automatic generation of the UI components of the PAIS by extracting role-specific process fragments from process models and transforming them into related UI components (cf. Section 4).

3.2 Elementary Transformation Patterns

Elementary transformation patterns (ETP) enable the transformation of single process model elements to simple UI elements. For example, an activity may be transformed into a simple user form. Thereby, the respective ETP considers activity input/output data elements and maps them to form elements.

Each transformation pattern has a unique *pattern name*. Its *description* summarizes the actions required to transform the process elements to corresponding UI elements. Furthermore, a real-world application *example* illustrates the application of the pattern. In this context, process elements are represented using BPMN and UI elements are illustrated in terms of screen mock-ups. The *problem* description refers to situations in which the patterns may be applied. The *implementation* defines implementation-specific requirements to be met to realize the pattern in a PAIS. *Related patterns* refers to related transformation patterns, e.g., patterns to be co-applied with the current pattern.

ETP1 (Human Activity Transformation). This pattern describes the elementary transformation of a single activity to a *Form Group Element (FGE)* (cf. Table 1); i.e., for each human activity of a process model, an FGE is generated. In modern PAIS, usually, such an FGE is represented by a dialog window.

ETP2 (Service Activity Transformation). The counterpart of *ETP1* is given by pattern *ETP2* (cf. Table 1). ETP2 creates application stubs for automated tasks not performed by a user; i.e., no user interaction element is created, but a stub for integrating application code executed by the PAIS. In BPMN,

respective activities are denoted as *service activities*. For example, a service activity could fetch data from a database, which is then displayed to the human user. Pattern ETP2 is needed to automatically generate complete UI components enabling interactions with both users and backend systems (cf. Section 3.3).

ETP1: Human Activity Transformation	
Description:	A *human activity* of a business process model (i.e., an activity to be performed by a human resource) is transformed into a *Form Group Element (FGE)*. An FGE corresponds to a UI element that contains UI elements for displaying or editing data.
Example:	A clerk must perform an activity, in which customer data is edited. 
Problem:	To perform a human activity within a PAIS, a user interaction is required.
Implementation:	An FGE can be implemented in terms of a dialog window. In the context of CTPs, an FGE constitutes a grouping element of the UI.
Related Pattern:	ETP2

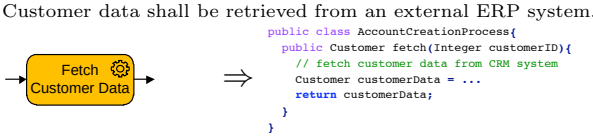
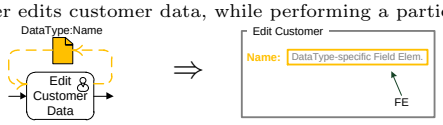
ETP2: Service Activity Transformation	
Description:	A *service activity* is transformed into an application stub executable by the PAIS. No human interaction is required.
Example:	Customer data shall be retrieved from an external ERP system. 
Problem:	Complex UI components not only require direct user interactions, but also interactions with backend information systems. Hence, service activities need to be integrated with the UI components as well.
Implementation:	For service activities, application stubs need to be created. Application components then may be provided by the process implementer.
Related Patterns:	ETP1

Table 1. Pattern Descriptions for Patterns ETP1 and ETP2

ETP3 (Data Flow Transformation). This elementary pattern allows transforming data elements and associated data flow edges to respective UI elements (cf. Table 2). Pattern *ETP3* as well as related patterns *ETP3.1-ETP3.4* generate *Field Elements (FE)* within an FGE; i.e., when generating the FGE (cf. ETP1), the data elements and edges of a process activity are transformed to input/output field elements of a UI component. In this context, patterns *ETP3.1* and *ETP3.2* are applied to indicate whether the respective field element is read-only or editable (with optionally pre-filled value). Further, *ETP3.3* transforms the data type of a data element to the FE; e.g., a boolean data element is transformed to a radio button element with two choices. Finally, *ETP3.4* allows transforming a business object to a UI component built upon *ETP3.3*.

ETP3.1 (WRITE Data Flow Transformation). *ETP3.1* transforms the *write access of a process activity to a data element* to an editable form element (cf. Table 2). The data type of the form element is determined by *ETP3.3*. Overall, *ETP3.1* ensures that the respective FE is editable and the user provides data, which is then mapped to the corresponding data element of the process model at run-time.

ETP3: Data Flow Transformation	
Description:	UI components are used to display and edit data elements of the process model. Pattern ETP3 as well as patterns ETP3.1-ETP3.4 transform these data elements and associated data edges to Field Elements (FE).
Example:	A user edits customer data, while performing a particular activity. 
Problem:	Human activities and associated data elements require UI field elements allowing the user to fill in or update information.
Implementation:	FGEs generated by ETP1 have to be enriched with FEs. This is done through the application of the elementary patterns ETP3.1-3.4.
Related Patterns:	ETP1, ETP3.1-ETP3.4

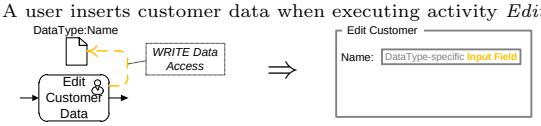
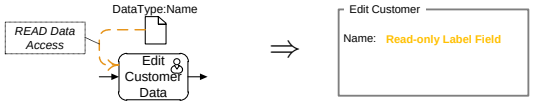
ETP3.1: WRITE Data Access Transformation	
Description:	A WRITE data access of an activity to a data element requires an input field element (FE) placed on the FGE and allowing the user to enter data. The value entered in the FE is then written to the respective data element.
Example:	A user inserts customer data when executing activity *Edit Customer*. 
Problem:	Inserting data values through UI components is crucial for users interacting with the PAIS.
Implementation:	When generating a FE, methods are required to forward the data entered by the user to the corresponding data element in the PAIS. The respective FE must be editable.
Related Patterns:	ETP1, ETP3

Table 2. Pattern Descriptions for Patterns ETP3 and ETP3.1

ETP3.2 (READ Data Transformation). *ETP3.2* transforms the *read access of an activity to a data element* to a read-only form element (e.g., a non-editable text label, cf. Table 3). At run-time, the FE is pre-filled with the value of the process data element.

In certain cases, an activity has both incoming and outgoing data edges linking it with the same data element; i.e., the activity may read and update the respective data element. Hence, both patterns *ETP3.1* and *ETP3.2* are applied. As a result, a pre-filled FE is obtained whose value may be updated when executing the activity and UI component respectively.

ETP3.3 (Primitive Data Type Transformation). This pattern transforms the primitive data types *integer*, *float*, *boolean*, *string*, *date*, and *URI* to typed form elements (cf. Table 3). For example, type *date* is transformed to an input field with a date picker validating the input. Overall, the most relevant primitive data types used in process models are covered [11]. Additionally, the respective form element is added to the FGE of the corresponding process activity.

ETP3.2: READ Data Access Transformation	
Description:	A READ data access creates a read-only field element (FE) in the UI, displaying the value of the respective process data element to the user.
Example:	Customer data is displayed to a clerk in his UI component. 
Problem:	For displaying process data elements, read-only FEs within the UI component are required.
Implementation:	Outgoing data edges of an activity are transformed to read-only FEs, which are pre-filled with the values of the respective data elements at run-time.
Related Patterns:	ETP1, ETP3

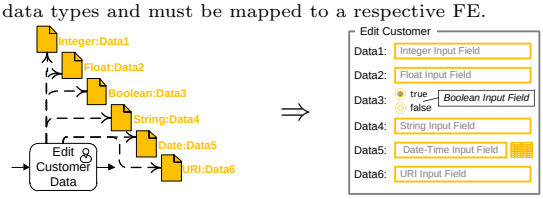
ETP3.3: Primitive Data Type Transformation	
Description:	Primitive data types require the generation of specific UI form elements in a consistent and comprehensible style.
Example:	A form for editing the business object *customer data* requires the processing of data elements like *customer number* or *birth date*. These have primitive data types and must be mapped to a respective FE. 
Problem:	For process data elements accessed by an activity, different form elements, depending on the respective primitive data types, have to be generated.
Implementation:	Each data element, having a primitive data type, is transformed to a specific FE and placed within the FGE of the associated activity.
Related Patterns:	ETP1, ETP3

Table 3. Pattern Descriptions for Patterns ETP3.2 and ETP3.3

ETP3.4 (Business Object Data Type Transformation). To handle complex data types (i.e., business objects), pattern *ETP3.4* can be applied (cf. Table 4) [12]. It deals with data assembled of primitive or other complex data types. For example, a customer business object may consist of the primitive data elements *name*, *street*, and *birth date*. The implementation of pattern ETP3.4 creates form group elements for logically grouping the primitive data elements the business object consists of.

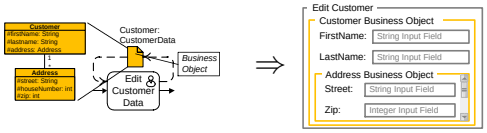
ETP3.4: Business Object Data Type Transformation	
Description:	A business object is transformed to an FGE. This FGE contains subordinate FEs for the data elements (with primitive data types) the business object consists of. For each primitive data type, *ETP3.3* is applied. If the business object refers to other business objects, *ETP3.3* is applied recursively.
Example:	A *customer* business object contains an s *address* business object that, in turn, consists of primitive data type attributes. 
Problem:	Business objects may comprise primitive data elements as well as business objects (complex data elements). All these data elements must be extracted and respective UI elements be generated for them.
Implementation:	Before generating FEs, all primitive data elements of a business object have to be extracted recursively and grouped according to the schema of the business object.
Related Patterns:	ETP3, ETP3.3

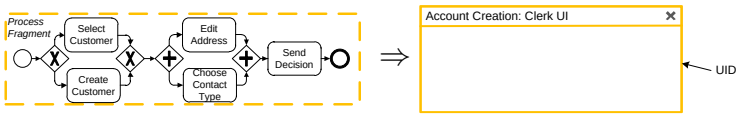
Table 4. ETP3.4: Business Object Data Type Transformation

3.3 Complex Transformation Patterns

Complex Transformation Patterns (CTPs) allow transforming entire process fragments of a process model, whose activities shall be performed by the same user, to corresponding UI components. When creating such a UI component both the control and data flow of the respective process fragment are considered. Hence, each of the generated UI components covers parts of the overall process logic. Furthermore, by combining *role-specific* activities in the same UI component, unnecessary UI context switches can be avoided. To structure such a UI component, tab elements—called *Tab Container Elements (TCE)*—are introduced. Moreover, a CTP interconnects TCEs according to the control flow of the process fragment to which it is applied. In this context, single activities and related data elements of the process fragment are transformed using ETPs.

CTP1 (Process Model Transformation). For a process fragment whose activities shall be processed by the same user or user role, pattern CTP1 generates a *User Interface Dialog (UID)* (cf. Table 5). Such a UID is a toplevel container and is represented by a dialog window in the PAIS, which contains a number of UI elements representing the activities and data elements.

CTP2 (Sequence Block Transformation). This pattern deals with the transformation of a sequence of activities (and SESE blocks respectively) to TCEs (cf. Table 5). For each activity (or SESE block) of the sequence, a TCE element is generated and linked to other TCEs according to the given activity sequence.

CTP1: Process Model Transformation	
Description:	For a particular process fragment, a surrounding *User Interface Dialog (UID)*, i.e., a toplevel container window, is generated. Following this, all other UI elements related to activities of this fragment are generated based on ETPs and CTPs, and are then embedded in the UID.
Example:	All interactions with a clerk shall be done using the same UI component.
	
Problem:	The UI elements related to the activities and data elements of a particular process fragment need to be mapped to a toplevel container window. The UI flow logic (e.g., the ordering in which field elements may be displayed or written) corresponds to the control flow of the given process fragment.
Implementation:	For each process fragment, a UID element (dialog window) is generated.
Related Patterns:	None

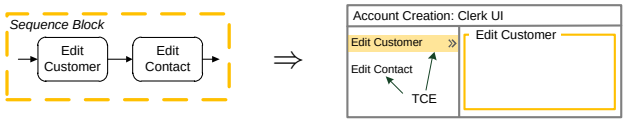
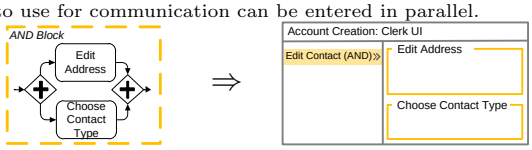
CTP2: Sequence Block Transformation	
Description:	A sequence of activities (and SESE blocks respectively) is transformed into a sequence of *Tab Container Elements (TCE)* to be processed in the same sequential order.
Example:	A clerk first edits the customer data. Afterwards, he edits the corresponding contact data.
	
Problem:	Human activities, performed in sequence by the same user (role), shall be accomplished using the same UI component, instead of using separate UI components (e.g., dialog windows) for each activity.
Implementation:	For each activity (or SESE block), a TCE element is created. The order in which these TCEs may be processed, corresponds to the one of the respective activities.
Related Patterns:	ETP1, ETP2, ETP3, CTP3

Table 5. Pattern Descriptions for Patterns CTP1 and CTP2

CTP3 (Parallel Block Transformation). This pattern transforms parallel activities (or SESE blocks) of a process fragment to respective UI elements within the same UID. The UI elements may then be accessed concurrently (cf. Table 6). The resulting UI component is similar to the one of a single activity (cf. ETP1, Table 1). However, pattern CTP3 not only covers the transformation of activities or SESE blocks arranged in parallel. It also enables the concurrent processing of arbitrary SESE blocks arranged in parallel to the respective UI elements.

CTP4 (XOR Block Transformation). This pattern is applied for transforming an XOR branching of a given process fragment to the UI component (cf. Table 6). CTP4 generates independent TCEs for each branch of the XOR branching. The decision, which branch and hence which TCE shall be selected, is made during run-time based on process data; e.g., whether the TCE element for creating a new customer or the one for editing an existing customer shall be displayed, depends on decision data that only becomes available during process

execution. In particular, run-time data for deciding which of the branches of an XOR branching shall be executed, is required.

CTP3: Parallel Block Transformation	
Description:	A parallel block and its activities are mapped to a single TCE for their processing. This TCE allows for their concurrent.
Example:	While editing the address of a customer, the contact type the customer wants to use for communication can be entered in parallel. 
Problem:	Activities (or SESE blocks) of a process fragment, which are performed in parallel by the same user (role), shall be mapped to the same UI component; UI elements then must be displayable/editable concurrently.
Implementation:	When applying CTP3, for each parallel branch, FGEs are added to the TCE.
Related Patterns:	ETP1, ETP2, ETP3, CTP2

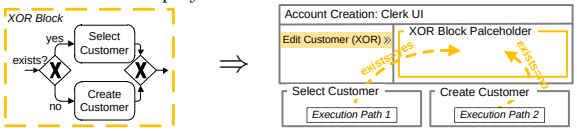
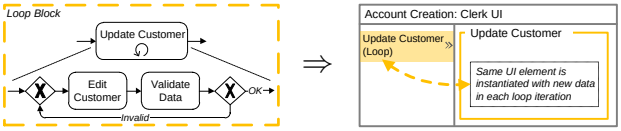
CTP4: XOR Block Transformation	
Description:	An XOR block and its branches are usually mapped to multiple UI components in the PAIS, one for each branch. By contrast, CTP4 transforms them to one TCE, if the corresponding activities are performed by the same user role. At run-time, this TCE then only displays the UI elements corresponding to the activities (or SESE blocks) of the selected branch.
Example:	If a customer entry already exists in the customer database, the form for editing customer data is displayed. Otherwise, the form for creating a new customer is displayed. 
Problem:	Branches of an XOR branching require the generation of several UI elements for each branch. At run-time, only the UI elements of the selected branch shall be displayed to the user.
Implementation:	A placeholder TCE is inserted at build-time. At run-time, the decision is made which concrete UI elements are dynamically added to that TCE.
Related Patterns:	ETP1, ETP2, ETP3, CTP2

Table 6. Pattern Descriptions for Patterns CTP3 and CTP4

CTP5 (Loop Block Transformation). This pattern transforms a loop block to elements of a UI component (cf. Table 7). For each loop, CTP5 generates a TCE as well as corresponding UI elements for activities (or SESE blocks) nested in this loop (cf. CTP1, Table 5). Additionally, a decision element is required to decide whether to leave the loop after completion of a particular iteration or trigger the next loop iteration. The iteration can be triggered by data elements processed during loop execution (e.g., evaluating input data for validity) or by external criteria (e.g., repeat calling someone until getting an answer).

CTP6 (Background Activity Transformation). Table 7 shows an advanced scenario for transforming process fragments to UI components. *CTP6* is a pattern reflecting the need for dynamically loading data elements by a service activity. More precisely, data has to be fetched from or stored to a backend system, while the user concurrently works on human activities by a user.
The next section introduces our overall UI transformation method. It applies the introduced transformation patterns in connection with process views [13].

CTP5: Loop Block Transformation	
Description:	A loop block is transformed to a TCE which then may contains the UI elements corresponding to the activities located inside this loop block.
Example:	Customer data shall be first edited and then validated. Depending on the validation result, editing may have to be redone (e.g., if invalid data was entered).
	
Problem:	Loop blocks may consist of nested activities and SESE blocks. Each loop iteration allow the user (role) to edit previously entered data.
Implementation:	For each loop block, a TCE is inserted. The latter is executed as long as the loop exit condition evaluates to false. Furthermore, all nested UI elements of the loop block are associated with the TCE.
Related Patterns:	ETP1, ETP2, ETP3, CTP2, CTP3, CTP4, CTP6

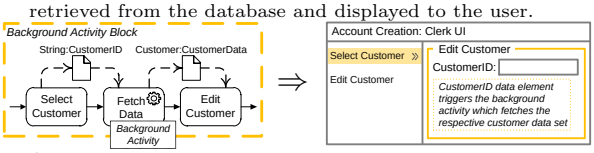
CTP6: Background Activity Transformation	
Description:	While human activities are performed by human resources, service activities may automatically fetch or save data concurrently in the background.
Example:	A user selects a customer name in order to edit respective customer data. After selecting the name, in the background, all available customer information is retrieved from the database and displayed to the user.
	
Problem:	A service activity needs to be executed concurrently to human activities performed by the same user (role). This requires the concurrent fetching/storing of data from a database as well as the automated and dynamic displaying of new form elements.
Implementation:	Dynamic forms, which contain background activities, need a change listener mechanism to detect user inputs and to react on them.
Related Patterns:	ETP1, ETP2, CTP2

Table 7. Pattern Descriptions for Patterns CTP5 and CTP6

4 Transforming Process Models to User Interfaces

Section 4 shows how the presented patterns can be used to transform process fragments into sophisticated UI components of the PAIS. In particular, we introduce a transformation method (cf. Section 4.1) and discuss how process fragments can be adapted through changes of the UI components (cf. Section 4.2).

4.1 User Interface Transformation Method

To transform a complete business process model, consisting of several process fragments, to multiple UI components, we introduced five steps (cf. Figure 3). Thereby, the number of generated UI components depends on the number of different users and user roles involved in the process.

Step 1. *Role-specific process views* [14,13,15] are created for the given process model. To be more precise, a process view abstracts from certain aspects of the process model. For example, it may only represent the activities of a particular user role or organisational unit. In our context, a role-specific process view constitutes the basis for creating a role-specific UI component.

Step 2. For each process view, a *User Interface Dialog (UID)* is created. As aforementioned, a UID acts as a toplevel container that includes all UI elements required for processing the activities of a specific process view. For this purpose, pattern CTP1 is applied (cf. Table 5).

Step 3. In order to transform complete process fragments to UI elements, complex transformation patterns CTP2-6 are applied. For each CTP applied, a *Tab Container Element (TCE)* is generated. Each TCE is represented in the tab bar area (cf. Figure 3, Step 3). When clicking on an item in this area, the corresponding UI elements are displayed. If there are nested SESE blocks, they are displayed in a hierarchical tree in the tab bar area.

Step 4. Single activities (ETP1+2) are transformed into *Form Group Elements (FGE)*. Basically, each FGE represents one activity in the process model. In case of a parallel branching, multiple activities are displayed on a TCE element in the UI (cf. Figure 3, Step 4).

Step 5. All data elements of the process view are transformed to *Field Elements (FE)*. These are layouted and positioned within an FGE. In this context, there exist different kinds of FEs depending on the data type of the respective data element (cf. ETP3.3, Table 3).

The structure of the resulting UI component is represented through a *User Interface Model (UIM)*. Figure 4b shows an example of such a UIM. This tree-based

schema describes the hierarchical structure of the UI as generated by our transformation method. Section 4.2 shows that this UIM can be also used to propagate changes of the UI component to the process model and vice versa.

4.2 Synchronizing Process Model and UI Changes

After generating complex UI components for a process model through process views and deploying them in the PAIS, users may want to modify the UI. For example, they might add a new FE or re-position elements within the UI.

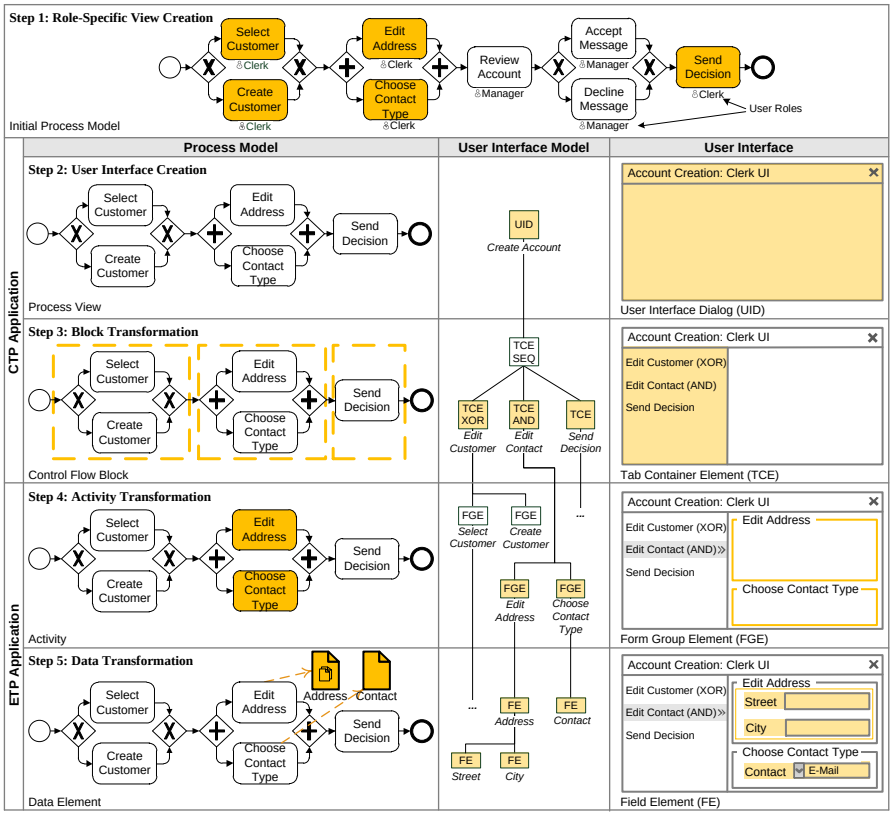


Fig. 3. Transformation Method

Basically, two categories of UI changes can be distinguished. *Local changes* are changes not affecting the associated process view. For example, assume that a user re-positions the FGE *Edit Address* within the TCE *Edit Contact (AND)* in Figure 4a. Such change would not affect the execution order of the activities

in the corresponding process view, i.e., it only affects the visual representation of the UI component. Hence, the change needs not be propagated to the view. By contrast, *global changes* modify the logic of the UI as well as the associated process view. For example, the user may want to add the FGE *Edit Phone Number* together with the respective FE *Phone* (cf. Figure 4a). This change then affects the control flow of the underlying process view as well. The correct position of the change within the UIM can be determined by the hierarchical structure of the GUI (cf. Figure 4b). The changes of the UIM are then propagated to the respective process view; note that the latter is represented by the UIM. Finally, the change of the process view has to be propagated to the basis process model on which the view is created. For this propagation the concepts developed in the *proView*[1] project can be applied [16,17].

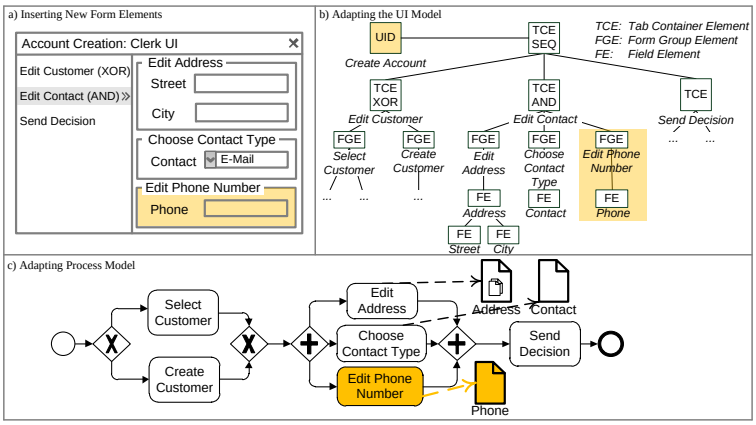


Fig. 4. Adapting Process Models through Changes in the UI Component

5 Proof-of-Concept Prototype

We developed a proof-of-concept prototype for the described UI generation approach. In particular, we implemented the component enabling process view generation as well as the described transformation patterns and method. This UI generation prototype has been realized in the context of the *proView* research project, which tries to reduce process model complexity for end-users by generating personalized and tailored views on process models for them [13,14]. The prototypical implementation is based on a layered REST architecture using the *vaadin* MVC web framework (cf. Figure 5). To evaluate the general validity of the transformation patterns and method, we used the prototype to generate

[1] http://www.dbis.info/proView

UI components for process models we gathered in several PAIS projects. This evaluation has shown remarkable results in the sense that the UI components generated by our approach are more or less the same as those manually implemented in respective projects. Hence, both implementation and maintenance effort in PAIS development can be significantly reduced.

Our evaluation has further shown that for sequences comprising numerous activities, the generated UI component may become too complex. In this case, the tab bar located on the left side of the UI contains too many entries. Here, additional transformation patterns need to be identified. Finally, run-time issues have not been fully considered yet. For example, it must be specified what shall happen if a user wants to step back in the UI, i.e., derived from the defined form logic.

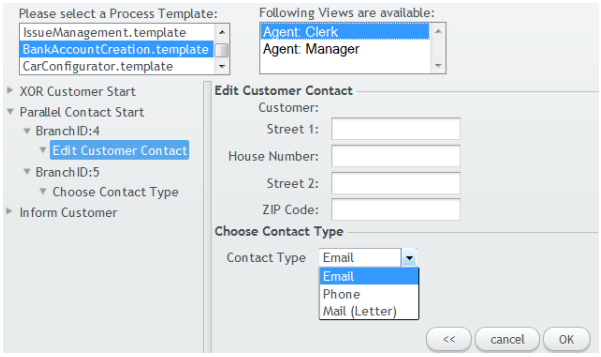


Fig. 5. User Form Generated by the Proof-of-Concept Prototype

6 Related Work

Generally, UI development is a time consuming and expensive task in the context of PAIS development [18].

Task Models: Task models describe the actions to be performed by a user when interacting with an information system and the respective UI to reach a specific goal. Different variants of task models exist: *Goals, Operators, Methods and Selection rules (GOMS)* [19], or *Hierarchical Task Analysis (HTA)* [20]. These approaches describe the goals, steps and operations of a user interface. *Concurrent Task Trees (CTT)*, in turn, provide a hierarchical task model supporting various types of tasks (e.g., automatic vs. manual task) and relationships between tasks (e.g., sequential vs. parallel execution) [21,22]. An overview of task

modeling approaches is provided in [23].

UI Description Language: As opposed to task models, UI description languages describe the concrete elements a user interface consists of, instead of their behaviour. Nevertheless, some of these languages include basic task modeling concepts as well [8]. Furthermore, respective languages are independent from a specific programming language and enable software developers to describe a user interface once, but deploy it on various devices (e.g., PC and tablets). Examples of UI languages include: *User Interface Markup Language (UIML)* [24], *Abstract User Interface Markup Language (AUIML)* [25], and *User Interface Extensible Markup Language (UsiXML)* [26]. Based on these languages, UI descriptions can be created at both built- and run-time.

Model-Driven UI Development: Model-driven UI development applies the principles of Model-Driven Development (MDD) to UI development. Although a lot of competing approaches exist, an accepted standard for model-driven UI development is missing [8,27,28,29,30]. FlowiXML [8], for example, provides a methodology to develop UIs for business processes, taking the organizational structure as well as the process model into account. FlowiXML suggests an eight-step-approach based on various model types (e.g.: CTT, Petri Nets, and UsiXML). However, it does not allow for the automated generation of UIs. Based on FlowiXML, [9] introduces the *Business Alignment Framework*. This four-step-approach provides task models (i.e., CTT) for user tasks with in the process model. Furthermore, a domain model supplements the task model. Based on these models, an abstract UI description is generated and transformed into a concrete UI component during process execution. This approach allows for changes based on UIs and discuss how to manually align them with process models. However, automatic propagation is not supported.
The approach provided in [31] transforms a process model into a human interaction perspective, which allows UI designers to specify data elements, user roles, tasks, and UI layout. This perspective enables detailed views on the process model. After manually refining them, corresponding UIs are generated during run-time. Furthermore, data-centered process management approaches offer a different (data-centered) view on business processes. Therefore, state transitions of process-related data elements are described. Based on this information, UIs can be generated as well [10,32].

UI Generation in existing PAIS: Contemporary PAIS are able to create certain UIs automatically (e.g., IBM WebSphere Lombardi [33]). To be more precise, single activities of a process model can be transformed into simple user forms, taking associated data elements into account similar to the ETPs. However, more complex scenarios are not covered. None of the presented approaches allows for the automatic generation of complex user interfaces based on process models. Finally, the adaption of process models based on changes of the corresponding UI is only considered rudimentarily.

7 Conclusion

In this paper, we showed how UI components can be automatically created from entire process fragments and process models respectively. For this purpose, elementary and complex transformation patterns were identified and described. Furthermore, a transformation method, which applies these patterns to create complex UIs based on process views, were introduced. Our approach further enables the propagation of UI changes (e.g., adding new input fields) to the associated process model and vice versa. Finally, we implemented our UI generation approach in a powerful proof-of-concept prototype. In summary, our approach will contribute to reduce costs for PAIS development and maintenance.
In future research, we will address the execution aspects of process models and associated UIs as well. In this context, features such as jumping back to an already edited UI element will be supported by adapting the corresponding process instance to assure a valid history of its execution. Further, we started to implement a UI editor which uses a set of predefined UI widgets and templates for creating sophisticated UI components for process models. In this editor, we use our transformation method and change propagation features for deriving complete process models from the created complex UIs.

References

1. Pradeep, H.: Process-User Interface Alignment: New Value From a New Level of Alignment. Align Journal (October 3, 2007)
2. Reichert, M., Weber, B.: Enabling Flexibility in Process-aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
3. Weber, B., Reichert, M., Mendling, J., Reijers, H.A.: Refactoring Large Process Model Repositories. Computers in Industry **62**(5) (2011) 467–486
4. Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Inf. Sys. **10**(2) (1998) 93–129
5. La Rosa, M., Wohed, P., Mendling, J., ter Hofstede, A.H.M., Reijers, H.A., van der Aalst, W.M.P.: Managing Process Model Complexity Via Abstract Syntax Modifications. IEEE Transactions on Industrial Informatics **7**(4) (2011) 614–629
6. Mendling, J., Strembeck, M.: Influence Factors of Understanding Business Process Models. In: Proc. BIS'08. (2008) 142–153
7. Dadam, P., Reichert, M.: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support. Computer Science - Research and Development **23**(2) (April 2009) 81–97
8. Garcia, J.G., Vanderdonckt, J., Calleros, J.M.G.: FlowiXML: A Step Towards Designing Workflow Management Systems. Int'l Journal of Web Engineering and Technology **4**(2) (2008) 163–182
9. Sousa, K., Mendonça, H., Vanderdonckt, J., Rogier, E., Vandermeulen, J.: User Interface Derivation from Business Processes: A Model-Driven Approach for Organizational Engineering. In: Proc. ACM SAC'08. (2008) 553–560
10. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards A Framework for Object-Aware Process Management. Journal Software Maintenance and Evolution: Research & Practice **23**(4) (2011) 205–244

11. Russell, N., ter Hofstede, A.H.M., Edmond, D., Aalst, W.M.P.V.D.: Workflow Data Patterns: Identification , Representation and Tool Support. In: Proc. ER 2005. (2005) 353–368
12. Hongxin, A., Yusheng, X., Zhixin, M., Li, L.: Integrating User Interfaces by business Object States. In: Proc. ICISE'10. (2010) 2900–2903
13. Reichert, M., Kolb, J., Bobrik, R., Bauer, T.: Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: Proc. ACM SAC'12, Riva del Garda (Trento), Italy (2012)
14. Kolb, J., Reichert, M.: Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In: Proc. S-BPM ONE 2012, CCIS 284. (2012) 237–251
15. Bobrik, R., Reichert, M., Bauer, T.: View-Based Process Visualization. In: Proc. 5th Int'l Conf. on Business Process Management, Brisbane, Australia (2007) 88–95
16. Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for User-centered Adaption of Large Process Models. In: Proc. Intl. Conf. on Service Oriented Computing (ICSOC'12), Shanghai, China (2012) to appear
17. Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: Proc. of the Business Process Management 2012 Demonstration Track, Tallinn, Estonia (2012) to appear
18. Hussmann, H., Meixner, G., Zuehlke, D.: Model-Driven Development of Advanced User Interfaces. Springer Berlin Heidelberg (2011)
19. Card, S., Moran, T., Newell, A.: The Psychology of Human-Computer Interaction. CRC (1983)
20. Annett, J.: Hierarchical Task Analysis. CRC (2003)
21. Paternò, F., Mancini, C., Meniconi, S., Maria, V.S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: Proc. IFIP TC13 Int'l Conf. on Human-Computer Interaction. (1997) 362–369
22. Paternò, F.: ConcurTaskTrees : An Engineered Approach to Model-based Design of Interactive Systems. The Handbook of Analysis for Human Computer Interaction (1999) 1–18
23. Limbourg, Q., Vanderdonckt, J.: Comparing Task Models for User Interface Design. The Handbook of Task Analysis for Human-Computer Interaction **6** (2004)
24. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., Shuster, J.: UIML: An Appliance-Independent XML User Interface Language. Computer Networks **31**(11-16) (1999) 1695–1708
25. Azevedo, P., Merrick, R., Roberts, D.: OVID to AUIML-User-Oriented Interface Modelling. In: Proc. 1st TUPIS'00. (2000)
26. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L.: USIXML: A Language Supporting Multi-path Development of User Interfaces. In: Engineering Human Computer Interaction and Interactive Systems. (2005) 134–135
27. Puerta, A., Eriksson, H., Gennari, J.H., Musen, M.A.: Model-Based Automated Generation of User Interfaces. In: Proc. 12th National Conference on Artificial Intelligence. (1994) 471–477
28. Traetteberg, H., Molina, P.J.: Making Model-Based UI Design Practical: Usable and Open Methods and Tools. In: Proc. IUI'04. (2004) 376–377
29. Lu, X.: Model Driven Development of Complex User Interface. In: Proc. MoDELS 2007, Workshop on Model Driven Development of Advanced User Interfaces. (2007)
30. Puerta, A., Maulsby, D.: MOBI-D: A Model-based Development Environment for User-Centered Design. In: Proc. CHI'97. (1997) 4–5
31. Sukaviriya, N., Sinha, V.: User-Centered Design and Business Process Modeling: Cross Road in Rapid Prototyping Tools. In: Proc. INTERACT'07. (2007) 165–178

32. Künzle, V., Reichert, M.: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: Proc. 12th Int'l Working Conference on Business Process Modeling, Development and Support (BPMDS'11). (2011) 201–215
33. Yang, S., Sun, Y., Waterhouse, J., Lau, D., Al-Hamwy, T.: Modeling and Implementing a Business Process Using WebSphere Lombardi Edition 7.1. In: Proc. CASCON'10. (2010) 374–375